

zt1: gestione multe e permessi nella Zona a Traffico Limitato

Progetto del corso di LCS 2008/09

Indice

1	Introduzione	1
1.1	Materiale in linea	2
1.2	Struttura del progetto e tempi di consegna	2
1.3	Valutazione del progetto	2
2	Il progetto: zt1	3
3	Formato permessi e passaggi	4
4	Il server dei permessi	4
5	Il processo zt1	5
6	Protocollo di interazione	5
6.1	Formato dei messaggi	5
6.2	Messaggi da Client a Server	6
6.3	Messaggi da Server a Client	6
7	Lo script mailscript	6
8	Istruzioni	7
8.1	Materiale fornito dai docenti	7
8.2	Cosa devono fare gli studenti	7
9	Parti Opzionali	7
10	Codice e documentazione	8
10.1	Vincoli sul codice	8
10.2	Formato del codice	8
10.3	Relazione	9

1 Introduzione

Il corso di Laboratorio di Programmazione Concorrente e di Sistema (AA538 – 6 crediti) prevede lo svolgimento di un progetto individuale suddiviso in tre frammenti. Questo documento descrive la struttura complessiva del progetto e

dei vari frammenti che lo compongono.

Il progetto consiste nello sviluppo del software relativo a **zt1** (Zona a Traffico Limitato): un sistema che gestisce i dati relativi all'accesso alla Zona a Traffico Limitato di un ipotetico comune. Il software viene sviluppato e documentato utilizzando gli strumenti, le tecniche e le convenzioni presentati durante il corso.

1.1 Materiale in linea

Tutto il materiale relativo al corso può essere reperito sul sito Web:

<http://www.cli.di.unipi.it/doku/doku.php/lcs/lcs09/start>

Il sito verrà progressivamente aggiornato con le informazioni riguardanti il progetto (es. FAQ, suggerimenti, avvisi), sul ricevimento e simili. Il sito è un Wiki e gli studenti possono registrarsi per ricevere automaticamente gli aggiornamenti delle pagine che interessano maggiormente. In particolare consigliamo a tutti di registrarsi alla pagina delle 'FAQ' e degli 'avvisi urgenti'.

Eventuali chiarimenti possono essere richiesti consultando i docenti di LCS durante l'orario di ricevimento, le ore in laboratorio e/o per posta elettronica.

1.2 Struttura del progetto e tempi di consegna

Il progetto deve essere sviluppato dallo studente individualmente e può essere consegnato entro il 1 Febbraio 2010.

La consegna del progetto avviene *esclusivamente* attraverso il target **consegna** del **Makefile** contenuto nel kit di sviluppo del progetto. Eventualmente, una copia del tar creato dal target **consegna** può essere allegata ad un normale messaggio di posta elettronica. Le consegne sono seguite da un messaggio di conferma da parte del docente all'indirizzo di mail da cui la consegna è stata effettuata. Se la ricezione non viene confermata entro 3/4 giorni lavorativi, contattare il docente per e-mail.

*I progetti che non rispettano il formato o non consegnati con il target **consegna** non verranno accettati.*

La data ultima di consegna è il 01/02/2010. Dopo questa data gli studenti dovranno svolgere il nuovo progetto previsto per il corso 2009/10.

Inoltre, gli studenti che consegnano una versione sufficiente del progetto finale entro il 30 Giugno 2009 accumuleranno il terzo bonus di 2, che contribuisce al voto finale con le modalità illustrate nelle slide introduttive del corso (vedi sito).

1.3 Valutazione del progetto

Al progetto viene assegnato un punteggio da 0 a 26 di cui 6 punti esclusivamente determinati dalla *qualità della documentazione allegata*. La valutazione del progetto è effettuata in base ai seguenti criteri:

- motivazioni, originalità ed economicità delle scelte progettuali
- strutturazione del codice (suddivisione in moduli, uso di makefile e librerie etc.)

- efficienza e robustezza del software
- modalità di testing
- aderenza alle specifiche
- qualità del codice C e dei commenti
- chiarezza ed adeguatezza della relazione (vedi Sez. 10.3)

La prova orale tenderà a stabilire se lo studente è realmente l'autore del progetto consegnato e verterà su tutto il programma del corso. Il voto dell'orale (ancora da 0 a 26) fa media con la valutazione del progetto per delineare il voto finale. In particolare, l'orale comprenderà

- una discussione delle scelte implementative del progetto e dei frammenti
- l'impostazione e la scrittura di script bash e makefile
- l'impostazione e la scrittura di programmi C + PosiX non banali (sia sequenziali che concorrenti)
- domande su tutto il programma presentato durante il corso.

Casi particolari Gli studenti lavoratori iscritti alla laurea triennale in Informatica possono consegnare i due frammenti e il progetto finale in un'unica soluzione in qualsiasi momento dell'anno ed essere valutati con votazione da 0 a 30. In questo caso è necessaria la certificazione da consegnare al docente.

Gli studenti che svolgono il progetto per abbreviazioni delle nuove lauree specialistiche sono invitati a contattare il docente.

Gli studenti iscritti ai vecchi ordinamenti (nei quali il voto di Lab. 4 contribuiva al voto di SO) avranno assegnato un voto in trentesimi che verrà combinato con il voto del corso di SO corrispondente secondo modalità da richiedere ai due docenti coinvolti.

2 Il progetto: `ztl`

Lo scopo del progetto è lo sviluppo di un sistema per la gestione dei permessi e delle multe relative alla Zona a Traffico Limitato di un ipotetico comune. Il sistema è costituito da tre processi concorrenti:

- **passaggi**: che raccoglie le notifiche di passaggio dai varchi e li fornisce sullo standard output
- **permserver**: che mantiene tutti i permessi attivi e
- **ztl**: che legge da standard input le notifiche dei passaggi interagisce con **permserver** per stabilire quali costituiscono effettive infrazioni.

passaggi è un generatore casuale di passaggi fornito dai docenti. **permserver** e **ztl** sono due processi multithreaded che devono essere realizzati nel progetto didattico.

Inoltre, deve essere realizzato uno script bash (**mailscript**) che, dato un file di infrazioni genera le lettere di notifica come specificato in Sez. 7.

3 Formato permessi e passaggi

Un permesso per una data targa UUdddUU è rappresentato secondo il formato:

```
UUdddUU gg/mm/aaaa-hh:mm gg/mm/aaaa-hh:mm
```

in cui i primi 7 caratteri (U sta per uppercase e d sta per digit) rappresentano la targa del veicolo cui si riferisce il permesso e i restanti rappresentano la data/ora di inizio del permesso data/ora di fine del permesso rispettivamente.

Allo stesso modo una registrazione di un passaggio è della forma:

```
UUdddUU gg/mm/aaaa-hh:mm
```

dove sono rappresentate la targa del veicolo e la data/ora in cui il passaggio è stato registrato.

4 Il server dei permessi

`permserver` viene attivato da shell con il comando

```
$ permserver file_permessi
```

dove `file_permessi` è il path del file che contiene tutti i permessi attivi. I permessi sono rappresentati su file ciascuno con il formato in Sez. 3 e separati da newline `\n`.

Se `file_permessi` esiste viene aperto in lettura e caricato in un albero binario di ricerca. Ogni nodo dell'albero corrisponde ad una targa (che costituisce anche la chiave dell'albero binario). Ad ogni targa viene associata la lista degli intervalli di permesso attivi. La definizione precisa delle strutture dati e delle funzioni utilizzabili sulla struttura è definita nel file di README-1 del primo kit del progetto.

Dopo la creazione dell'albero viene attivato un thread *writer* che ogni NSEC¹ secondi controlla se il file `file_permessi` è stato modificato e (nel caso lo sia stato) ricrea l'albero con il contenuto del nuovo file.

All'attivazione, `permserver` crea anche una socket AF_UNIX

```
./tmp/permsoc
```

su cui i client apriranno le connessioni con il server per chiedere informazioni sui permessi. Il server è multithreaded in quanto ogni richiesta viene gestita da un thread attivato allo scopo.

Il server può essere terminato *gentilmente* inviando un segnale di `SIGTERM` o `SIGINT`. All'arrivo di uno di questi segnali il server deve eliminare la socket `permsoc` ed eventuali file temporanei, terminare i thread che lo compongono in modo che eventuali richieste pendenti da parte dei client vengano completate correttamente ed uscire.

Quando il server è attivo accetta dai client richieste di connessione e richieste di verifica di passaggi nella ZTL. Il protocollo di interazione è descritto nella Sezione 6.

¹NSEC *deve* essere definito come definito con una macro. Il test finale assume che il suo valore sia 3.

5 Il processo ztl

ztl è un comando Unix che una volta attivato legge dallo standard input i passaggi dai varchi ZTL con il formato descritto in Sez. 3 e scrive su un file di log eventuali infrazioni. Il nome del file di log viene fornito come argomento al momento dell'attivazione

```
$ ./ztl logfile
```

Appena attivato, il client effettua il parsing delle opzioni, e se il parsing è corretto cerca di collegarsi con il server dei permessi (su `./tmp/permssock`) per un massimo di 5 tentativi a distanza di 1 secondo l'uno dell'altro. Se il collegamento ha successo crea il file di log **logfile** ed inizia a leggere i passaggi dallo standard input interagendo con **permserver** per verificarli.

Processo **ztl** è costituito da più thread paralleli:

- *dispatcher* : il thread che si occupa di leggere i passaggi da standard input e di smistarli ai thread worker per la verifica
- *worker* : il gruppo di thread (uno per ogni richiesta) che interagisce con **permserver** per la verifica
- *writer* : il thread che si occupa di scrivere sul logfile.

I worker inseriscono le infrazioni in una struttura dati *S* condivisa con il writer. Il paradigma di interazione è quello produttore-consumatore. Lo studente deve scegliere la struttura dati più adeguata per implementare *S* e motivare la sua scelta all'interno della relazione finale.

Le multe sono registrate su file nel formato specificato in Sec. 3 separati da newline `\n`.

Il processo può essere terminato *gentilmente* inviando un segnale di **SIGTERM** o **SIGINT**. All'arrivo di questo segnale il processo deve terminare la verifica dei passaggi già letti da stdin, scrivere sul file di log eventuali multe ancora presenti in *S* e ripulire l'ambiente da eventuali file temporanei.

6 Protocollo di interazione

Il server dei permessi e **ztl** interagiscono utilizzando *socket AF_INET*.

I client si connettono al server attraverso la socket `./tmp/permssock`, creata all'avvio del server².

6.1 Formato dei messaggi

I messaggi scambiati fra server e client hanno la seguente struttura:

```
typedef struct {
    char type;
    unsigned int length;
    char* buffer;
} message_t;
```

²Sarebbe più logico creare la socket nella directory `/tmp/` di sistema ma per non avere problemi di interazioni indesiderate fra progetti diversi sulle macchine del cli è meglio creare tutte le pipe in una directory `./tmp/` locale alla directory di progetto

Il campo **type** è un **char** (8 bit) che contiene il tipo del messaggio spedito. **type** può assumere i seguenti valori:

```
#define MSG_ERROR      'E'
#define MSG_OK         'O'
#define MSG_NO         'N'
#define MSG_CHECK      'C'
```

Il campo **length** è un **unsigned int** che indica la dimensione del campo **buffer**. Se **buffer** è una stringa il suo valore comprende anche il terminatore di stringa **'\0'** finale che deve essere presente nel **buffer**. Il campo **length** vale 0 nel caso in cui il campo **buffer** non sia significativo. Il campo **buffer** è un puntatore a un **buffer** di caratteri.

6.2 Messaggi da Client a Server

Nei messaggi spediti dal client **zt1** al Server, il campo **type** può assumere solo il valore **MSG_CHECK**. In questo caso il **buffer** contiene il passaggio da verificare. Se il passaggio era autorizzato da un permesso, e quindi non costituisce infrazione il server risponde con **MSG_OK** altrimenti con **MSG_NO**. Se si è verificato un errore viene invece generato un **MSG_ERROR** ed in questo caso il campo **buffer** può riportare un messaggio di errore.

6.3 Messaggi da Server a Client

Nei messaggi spediti dal Server a Client, il campo **type** può assumere i seguenti valori:

MSG_ERROR Messaggio di errore. Questo tipo di messaggio viene spedito quando si riscontra un errore. Il campo **buffer** contiene la stringa che definisce l'errore riscontrato.

MSG_OK Segnala che il passaggio era permesso, non c'è stata infrazione.

MSG_NO Segnala che il passaggio non era permesso, quindi c'è stata infrazione.

7 Lo script mailscript

mailscript è uno script **bash** che elabora off-line il file di log generato dal processo **zt1** e compone le lettere di notifica della/e infrazioni. Lo script viene attivato come:

```
$ mailscript logfile anagrafe [ dir ]
```

dove **logfile** ed **anagrafe** sono due file di testo (**ASCII**) e **dir** (opzionale) è il nome di una directory dove memorizzare le lettere. **logfile** è il file che contiene tutti i passaggi dalla **ZTL** che costituiscono una infrazione (quello creato da **zt1**). **anagrafe** invece contiene le informazioni relative a nome cognome indirizzo dei proprietari delle auto, secondo il formato:

BU995PK, Del Ponte, Carlo, Pisa, Largo Pontecorvo, 5

quindi targa, cognome, nome, localita, indirizzo con campi separati da virgole.

Lo script deve controllare la validità dei suoi argomenti, scorrere il file `logfile` e per ogni targa generare una singola lettera indirizzata al proprietario che contenga la lista delle infrazioni rilevate. La lettera verrà salvata in un file con lo stesso nome della targa all'interno della directory `dir` (se specificata) o nella directory corrente.

L'idea è quella di utilizzare lo script a fine giornata per generare ed inviare le lettere relative a tutte le infrazioni rilevate.

Informazioni sul testo esatto della lettera da generare ed altri dettagli sono forniti nel README-2 del secondo frammento.

8 Istruzioni

8.1 Materiale fornito dai docenti

Nei kit del progetto vengono forniti

- funzioni di test e verifica
- `makefile` per test e consegna
- file di intestazione (`.h`) con definizione dei prototipi e delle strutture dati
- vari README di istruzioni

8.2 Cosa devono fare gli studenti

Gli studenti devono:

- leggere *attentamente* i README e capire il funzionamento del codice fornito dai docenti
- implementare le funzioni richieste e testarle
- verificare le funzioni con i test forniti dai docenti (attenzione: questi test vanno eseguiti su codice già corretto e funzionante altrimenti possono dare risultati fuorvianti o di difficile interpretazione)
- preparare la *documentazione*: vedi la Sez. 10.
- sottomettere i tre frammenti del progetto esclusivamente utilizzando il `makefile` fornito e seguendo le istruzioni nel README.

9 Parti Opzionali

Possono essere realizzate funzionalità ed opzioni in più rispetto a quelle richieste (ad esempio il bilanciamento dell'albero di ricerca del server, o un thread pool).

Le parti opzionali devono essere spiegate nella relazione e corredate di test appropriati.

10 Codice e documentazione

In questa sezione, vengono riportati alcuni requisiti del codice sviluppato e della relativa documentazione.

10.1 Vincoli sul codice

Makefile e codice devono compilare ed eseguire CORRETTAMENTE su (un sottinsieme non vuoto del) le macchine del CLI. Il README (o la relazione) deve specificare su quali macchine è possibile far girare correttamente il codice. Inoltre, se si usano software e librerie non presenti al CLI: (1) devono essere presenti nel tar TUTTI i file necessari per l'installazione in locale del/i tool e (2) devono essere presenti nel makefile degli opportuni target per effettuare automaticamente l'installazione in locale. Se questa condizione non è verificata il progetto non viene accettato per la correzione.

La stesura del codice deve osservare i seguenti vincoli:

- la compilazione del codice deve avvenire definendo delle regole appropriate nella parte iniziale del makefile contenuto nel kit;
- il codice deve compilare senza errori o warning utilizzando le opzioni `-Wall -pedantic`
- se non siamo sicuri della presenza del carattere terminatore di stringa NON devono essere utilizzate funzioni per la manipolazione delle stringhe che **non** limitano il numero di caratteri scritti/manipolati, ad esempio la `strcpy()` deve essere evitata a favore della `strncpy()` in cui è possibile fissare il massimo numero di caratteri copiati (per le motivazioni consultare il man in linea)
- NON devono essere utilizzate funzioni di temporizzazioni quali le `sleep()` o le `alarm()` per risolvere problemi di race condition o deadlock fra i processi/thread. Le soluzioni implementate devono necessariamente funzionare qualsiasi sia lo scheduling dei processi/thread coinvolti
- DEVONO essere usati dei nomi significativi per le costanti nel codice (con opportune `#define` o `enum`)

10.2 Formato del codice

Il codice sorgente deve adottare una convenzione di indentazione e commenti chiara e coerente. In particolare deve contenere

- una intestazione per ogni file che contiene: il nome ed il cognome dell'autore, la matricola, il nome del programma; dichiarazione che il programma è, in ogni sua parte, opera originale dell'autore; firma dell'autore.
- un commento all'inizio di ogni funzione che specifichi l'uso della funzione (in modo sintetico), l'algoritmo utilizzato (se significativo), il significato delle variabili passate come parametri, eventuali variabili globali utilizzate, effetti collaterali sui parametri passati per puntatore etc.
- un breve commento che spieghi il significato delle strutture dati e delle variabili globali (se esistono);

- un breve commento per i punti critici o che potrebbero risultare poco chiari alla lettura
- un breve commento all'atto della dichiarazione delle variabili locali, che spieghi l'uso che si intende farne

10.3 Relazione

La documentazione del progetto consiste nei commenti al codice e in una breve relazione (massimo 10 pagine) il cui scopo è quello di descrivere la struttura complessiva del lavoro svolto. La relazione *deve rendere comprensibile il lavoro svolto ad un estraneo, senza bisogno di leggere il codice se non per chiarire dettagli implementativi. In particolare NON devono essere ripetute le specifiche contenute in questo documento.* In pratica la relazione deve contenere:

- le principali scelte di progetto (strutture dati principali, algoritmi fondamentali e loro motivazioni)
- la strutturazione del codice (logica della divisione su più file, librerie etc.)
- la struttura dei programmi sviluppati
- la struttura dei programmi di test
- le difficoltà incontrate e le soluzioni adottate
- quanto altro si ritiene essenziale alla comprensione del lavoro svolto
- README di istruzioni su come compilare/eseguire ed utilizzare il codice

La relazione deve essere in formato PDF.